20MCA132 OBJECT ORIENTED PROGRAMMING LAB

Introduction

- Java programming language was originally developed by Sun Microsystems which was initiated by James Gosling and Patrick Naughton later acquired by Oracle Corporation.
- The language, initially called 'Oak' and later being renamed as Java, from a list of random words.
- Sun released the first public implementation as Java 1.0 in 1995.

Features of Java

- **Object Oriented:**
  - In Java, everything is an Object. Java can be easily extended since it is based on the Object model.
- **Compiled and Interpreted Language**
  - Java compiler translates source code into byte code.
  - Java interpreter generates machine code.
- **Simple:**
  - Java is designed to be easy to learn. If you understand the basic concept of OOP Java, it would be easy to master.
- **Platform Independent**:
  - Unlike many other programming languages including C and C++, when Java is compiled, it is not compiled into platform specific machine, rather into platform independent byte code.
  - This byte code is distributed over the web and interpreted by the Virtual Machine (JVM) on whichever platform it is being run on.
- **Architecture-neutral:**
  - Java compiler generates an architecture-neutral object file format, which makes the compiled code executable on many processors, with the presence of Java runtime system.
- **Portable:**

   - Being architecture-neutral and having no implementation dependent aspects of the specification makes Java portable.

- **Robust and secure**
   - Java is a most strong language which provides many securities to make certain reliable code.
   - It is design as garbage –collected language, which helps the programmers virtually from all memory management problems.
   - Java also includes the concept of exception handling, which detain serious errors and reduces all kind of threat of crashing the system.
   - Security is an important feature of Java and this is the strong reason that programmer use this language for programming on Internet.
   - The absence of pointers in Java ensures that programs cannot get right of entry to memory location without proper approval.
- **Distributed**
   - Java is called as Distributed language for construct applications on networks which can contribute both data and programs.
   - Java applications can open and access remote objects on Internet easily.
   - That means multiple programmers at multiple remote locations to work together on single task.

Java	**C++**
Java is true Object-oriented	C++ is basically C with Object-oriented language extension.
Java does not support operator overloading.	C++ supports operator overloading.
Java does not support multiple of inheritance classes. This is accomplished using a new feature called "interface".	C++ supports multiple inheritance of classes
Java does not support global variable. Every variable should declare in class.	C++ support global variable.

Java does not use pointer.	C++ uses pointer
It Strictly enforces an object-oriented programming	It Allows both procedural programming and object oriented programming paradigm.
There are no header files in Java.	We have to use header file in C++.

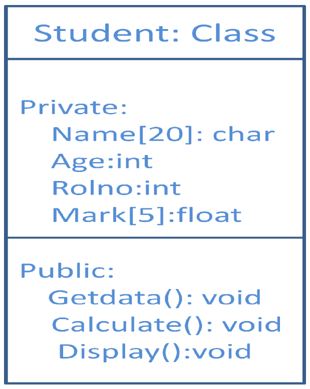
Concepts of OOP

Object
Class
Data abstraction
Data encapsulation
Inheritance
Polymorphism

- **Object**
    - Objects are important runtime entities in object oriented method.
    - For eg: a person, a bank account, etc.
    - Each object holds data and code to operate the data.
- **Class**
    - A class is a collection of objects of similar type
    - A class is a set of objects with similar properties (attributes), common behavior (operations), and common link to other objects.
    - Classes are user defined data types and work like the build in type of the programming language.
    - The objects are variable of type class.

Dynamic binding

Representation of a class using class diagram

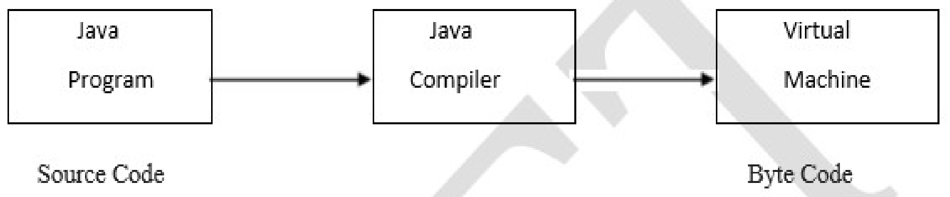


- **Data Encapsulation**
  - Data Encapsulation means wrapping of data and functions into a single unit (i.e. class).
  - It is most useful feature of class.
  - The data is not easy to get to the outside world and only those functions which are enclosed in the class can access it.
  - This insulation of data from direct access by the program is called as Data hiding.
- **Data Abstraction**
  - Data abstraction refers to the act of representing important description without including the background details or explanations.
- **Inheritance**
  - Inheritance is the process by which objects of one class can get the properties of objects of another class.
  - Inheritance means one class of objects inherits the data and behaviors from another class.
  - Inheritance provides the important feature of OOP that is reusability.
  - That means we can include additional characteristics to an existing class without modification.
  - This is possible deriving a new class from existing one.
- **Polymorphism**
  - (Poly means ―many‖ and morph means ―form‖).
  - Polymorphism means the ability to take more than one form.

  - Polymorphism plays a main role in allocate objects having different internal structures to share the same external interface.
  - Two types:- runtime polymorphism, compile time polymorphism.
- **Dynamic Binding**
  - Binding refers to the linking of a procedure call to the code to be executed in response to the call.
  - Dynamic binding means that the code related with a given procedure call is not known until the time of the call at run time.
  - Dynamic binding is associated polymorphism and inheritance.

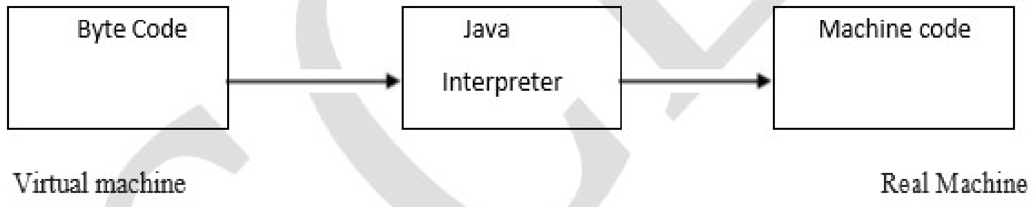
Java Virtual machine

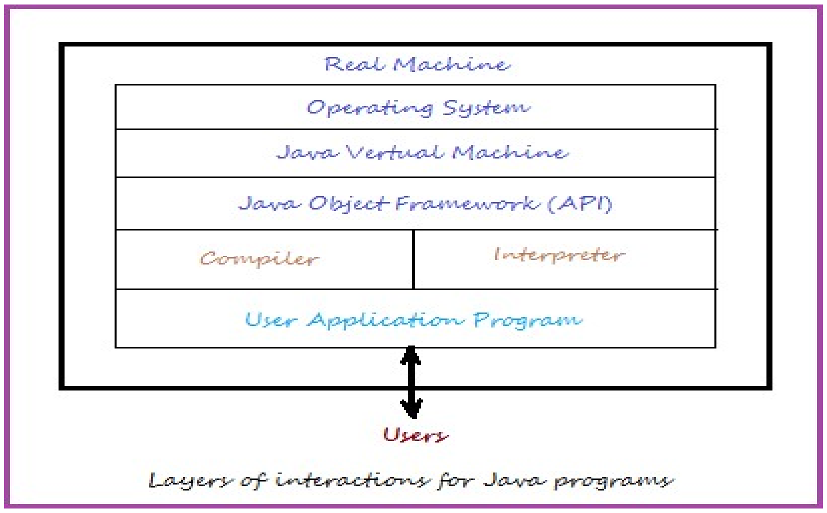
Java compiler convert the source code into Intermediate code is called as byte code. This machine is called the Java Virtual machine and it exits only inside the computer memory.



Process of compilation

The Virtual machine code is not machine specific. The machine specific code is generated by Java interpreter by acting as an intermediary between the virtual machine and real machines shown below



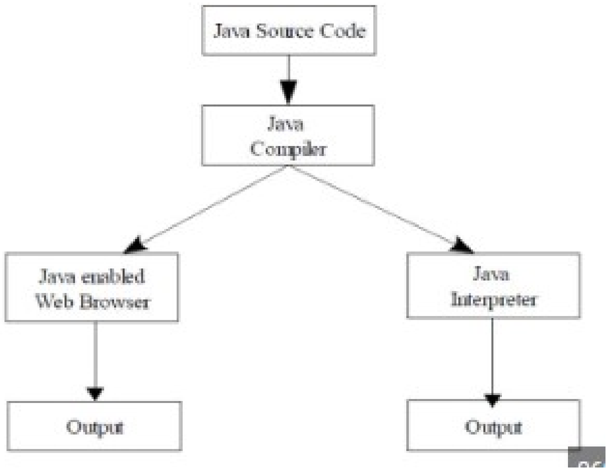


Types of Java Applications

There are mainly 2 types of applications that can be created using java programming.

◦ Standalone Application:- These are the programs written in java to carry out certain task on stand-alone local machine

◦ Web Application:- Web applets type programs used for creating internet applications



Java Environment

Java environment includes a large number of development tools and hundreds of classes & methods. The development tools are part of the system known as Java Development Kit (JDK).

- JDK is a collection of tools that are used for developing and running java programs.

The classes& methods are the part of the Java Standard Library (JSL) and also known as the Application Programming Interface (API).

Java Development Kit(JDK)

javac - The Java Compiler

java - The Java Interpreter

jdb- The Java Debugger

appletviewer -Tool to run the applets

javap - to print the Java bytecodes

javaprof - Java profiler

javadoc - documentation generator

javah - creates C header files

Application Programming Interface (API)

The Java API includes hundreds of classes and methods grouped into several functional packages. Some packages are

- Language support package:- a collection of classes required for implementing basic features of java.

  java.lang

- Input/output packages:- classes required for input/output manipulations

  java.io

- Utility packages:- class to provide utility functions such as date, time ……

  java.util

Popular Java Editors

To write your Java programs, you will need a text editor. These are known as IDE's (integrated development environment)

Notepad: On Windows machine, we can use any simple text editor like Notepad

Netbeans: A Java IDE that is open-source and free, which can be downloaded .

Eclipse: A Java IDE developed by the eclipse open-source community and can be downloaded.

Sample program

```
class sample
        {
                // This is my first java program.
                public static void main(String args[ ])
                {
                        System.out.println("Hello World");
                 }
        }
```

Follow the steps

- Open notepad and do the code as above.
- Save the file as: sample.java
- Open a command prompt window and go to the directory where we saved the program.
- **Set Path in Windows:**
  Open command prompt (cmd), go to the place where we have installed java on our system and locate the bin directory, copy the complete path and write it in the command like this.

  ```
  set path=C:\My_Folder\Java\jdk1.8.0_121\bin
  ```
- Compile the program as

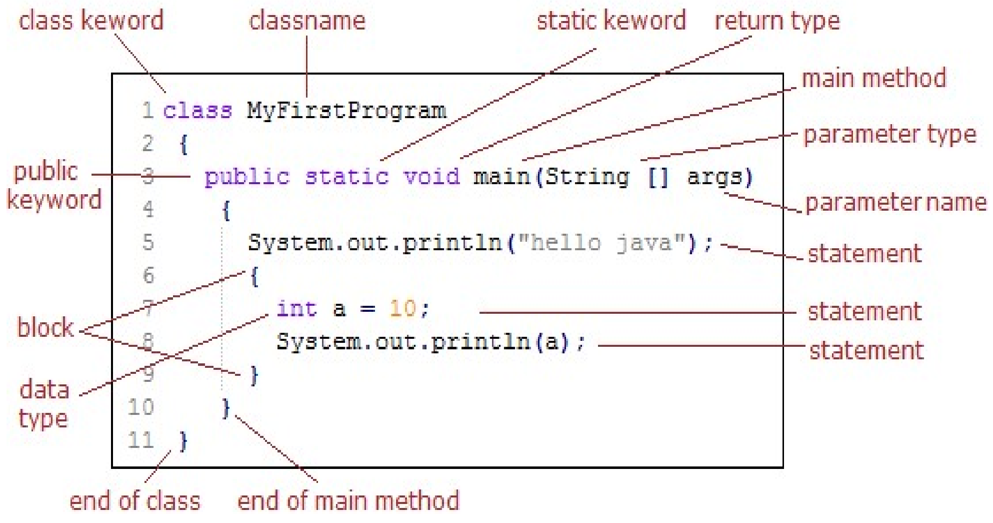
  ```
  javac sample.java
  ```
- Run the program as

  ```
  java sample
  ```
- **The output is**

  ```
  Hello World
  ```

- ☐ **class** keyword is used to declare a class in java.
- ☐ **public** keyword is an access modifier which represents visibility. It means it is visible to all.
- ☐ **static** is a keyword. If we declare any method as static, it is known as the static method. The core advantage of the static method is that there is no need to create an object to invoke the static method. The main method is executed by the JVM, so it doesn't require creating an object to invoke the main method. So it saves memory.
- ☐ **void** is the return type of the method. It means it doesn't return any value.
- ☐ **main** represents the starting point of the program.
- ☐ **String args[ ]** is used for command line argument.

- **System.out.println()** is used to print statement. Here, System is a class, println() is the method.



```
class MyFirstProgram
{
  public static void main(String [] args)
   {
     System.out.println("hello java");
     {
       int a = 10;
       System.out.println(a);
     }
   }
}
```

Java Tokens

A **token** is the smallest element of a program that is meaningful to the compiler. Tokens can be classified as follows:

Keywords
Identifiers
Constants
Special Symbols
Operators

Keyword:

- Keywords are pre-defined or reserved words in a programming language.
- Each keyword is meant to perform a specific function in a program.
- Eg: class, interface, import, int, float,………

Identifiers:

- Identifiers are used as the general terminology for naming of variables, functions and arrays.
- These are user-defined names.
- We cannot use keywords as identifiers; they are reserved for special use.

Constants/Literals:

- Constants values cannot be modified by the program once they are defined.
- Constants refer to fixed values.
- They are also called as literals.

Special Symbols:

- special symbols are used in Java having some special meaning and thus, cannot be used for some other purpose.
- **Eg: ( ), { }, [ ]……..**

Operators:

Java provides many types of operators which can be used according to the need.
Eg:

Arithmetic Operators - +, -, *, /, %

Unary Operators - ++, --

Assignment Operator - =

Relational Operators - <, >, <=, >=, ==, !=

Logical Operators - logical AND:- &&, logical OR:- ||

Ternary Operator - Conditional operator:- ?=

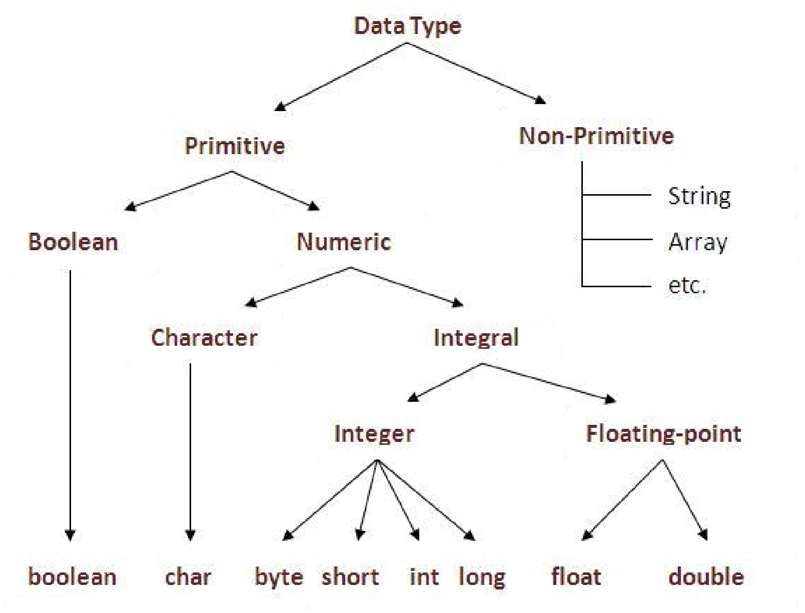
Bitwise Operators - bitwise AND:- &
bitwise exclusive OR :- ^
bitwise inclusive OR :- |

Shift Operators – left shift operator:- <<
Right shift operator:- >>

Data Types in Java

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

- **Primitive data types**
- **Non-primitive data types**



Java Variables

A variable is a container which holds the value while the Java program is executed. A variable is assigned with a data type. Variable is a name of memory location. There are three types of variables in java:

Local variable
instance variable
Static variable

1) Local Variable

A variable declared inside the body of the method is called local variable. We can use this variable only within that method and the other methods in the class aren't even aware that the variable exists. A local variable cannot be defined with "static" keyword.

2) Instance Variable

A variable declared inside the class but outside the body of the method, is called instance variable. It is called instance variable because its value is instance specific and is not shared among instances.

3) Static variable

A variable which is declared as static is called static variable. It cannot be local. We can create a single copy of static variable and share among all the instances of the class. Memory allocation for static variable happens only once when the class is loaded in the memory.

```
class A
{
        int data=50;//instance variable
        static int m=100;//static variable
        void method()
        {
            int n=90;//local variable
        }
}//end
```

Operators in Java

Example 1: Add Two Numbers

```
class Simple
{
            public static void main(String arg[ ])
            {
                int a=10;
                int b=10;
                int c=a+b;
                System.out.println("Sum="+c);
            }
}
```

Output
Sum=20

Example 2: Unary Operator

```
class OperatorExample
{
public static void main(String args[ ])
{
        int x=10;
        System.out.println(x++);
        System.out.println(++x);
        System.out.println(x--);
        System.out.println(--x);
        int b=10;
        System.out.println(b++ + ++b); //10+11=21
    } }
```

Output

```
10
12
 12
10
21
```

Shift operators

- ☐ Shifting bits can perform multiplication and division instead of using the multiplication and division operators.
- ☐ It turns out that moving bits is exceptionally fast, whereas multiplication and division are very slow.
- ☐ These operators are used to shift the bits of a number left or right thereby multiplying or dividing the number by two respectively.
- ☐ They can be used when we have to multiply or divide a number by two.
- ☐ General format:

  number **shift_op** number_of_bits;
- ❑ For eg:
  - ❑ 20<< 2;
  - ❑ 10>>4;

1. Shift Left Operator

- ☐ Shifting a value to the left (<<) results in multiplying the value by a power of two.
- ☐ The number_ of_ bits specifies how many bits to shift the value over; it has the effect shown in Table

number_ of_ bits	**Multiplication Result**
1	2
2	4
3	8
4	16
5	32

- Or stated more simply shifting a value, *n* bits, is the same as multiplying it by 2^n
- Example 1:

  a = 5;

  a<<1 = 10;
- Binary representation of 5 is 0000 0101
- **After the left shift by one bit is**

a << 1 = 0000 1010 = 10

- ☐ Example 2

  int i=10;

  int result = i << 2; // result = 10*4=40

- ▪ Binary representation of 10 is 0000 1010
- ▪ **After the left shift by two bits is**
- ▪ i << 2 =**1010 0000** = 40

Example 3: Left shift Operator

```
class shiftExample
{
        public static void main(String args[ ])
        {
        System.out.println(10<<2);//10*2^2=10*4=40
        System.out.println(10<<3);//10*2^3=10*8=80
        System.out.println(20<<2);//20*2^2=20*4=80
        System.out.println(15<<4);//15*2^4=15*16=240
        }
}
```

2. Shift Right Operator

- ☐ The Java right shift operator, >>, is used to move left operands value to right by the number of bits specified by the right operand.
- ☐ Shifting a value to the right (>>) is the same as dividing it by a power of two.
- ☐ Or stated more simply shifting right a value, *n* bits, is the same as dividing it by 2^n. For example:
  - ◘ int i = 20;
  - ◘ int result = i >> 2; // result = 20/(2^2)=5

Example 4: Right shift Operator

```
class OperatorExample
{
        public static void main(String args[ ])
        {
        System.out.println(10>>2);//10/2^2=10/4=2
        System.out.println(20>>2);//20/2^2=20/4=5
        System.out.println(20>>3);//20/2^3=20/8=2
        } }
```

Output:

```
2
5
2
```

Java AND Operators: Logical && and Bitwise &

- ☐ The logical && operator doesn't check second condition if first condition is false. It checks second condition only if first one is true.
- ☐ The bitwise & operator always checks both conditions whether first condition is true or false.

Example 5: AND Operators

```
class ANDExample
{
        public static void main(String args[])
        {
          int a=10;
          int b=5;
          int c=20;
          boolean r1= a<b && a<c;
          boolean r2= a<b & a<c;
          System.out.println("Result1=" + r1);//false && true = false
          System.out.println(" Resul2t=" + r2);//false & true = false
        } }
```

Output:

```
Result1=false
Result2=false
```

Example 6: AND Operators

```
class OperatorExample
{
                public static void main(String args[])
                {
                        int a=10;
                        int b=5;
                        int c=20;
                        System.out.println(a<b&&a++<c);//false && true = false
                        System.out.println(a);//10 because second condition is not checked
                        System.out.println(a<b&a++<c);//false && true = false
                        System.out.println(a);//11 because second condition is checked
                } }
```

Output:

```
false
10
false
11
```

Java OR Operator : Logical || and Bitwise |

- ☐ The logical || operator doesn't check second condition if first condition is true. It checks second condition only if first one is false.
- ☐ The bitwise | operator always checks both conditions whether first condition is true or false.

Example 7: OR Operators

```
class OROperator
{
        public static void main(String args[])
        {
        int a=10;
        int b=5;
        int c=20;
        System.out.println(a>b||a<c);//true || true = true
        System.out.println(a>b|a<c);//true | true = true
        System.out.println(a>b||a++<c);//true || true = true
        System.out.println(a);//10 because second condition is not checked
        System.out.println(a>b|a++<c);//true | true = true
        System.out.println(a);//11 because second condition is checked
        }
}
```

Output:

```
true
true
true
10
```

```
true
11
```

Java Ternary Operator

- Java Ternary operator is used as one liner replacement for if-then-else statement and used a lot in Java programming.
- It is the only conditional operator which takes three operands.

Example 8:

```
class ternaryExample
{
        public static void main(String args[])
        {
        int a=2, b=5;
        int min=(a<b)?a:b;
        System.out.println("Small="+min);
        a=10,  b=5;
        min=(a>b)?b:a;
        System.out.println("Small2="+min);
        }
}
```

Output:
Small=2
Small2=5

Java Assignment Operator

Example 9:

```
class OperatorExample
{
                public static void main(String[ ] args)
                {
                        int a=10;
                        a+=3;//10+3
                        System.out.println(a);
                        a-=4;//13-4
                        System.out.println(a);
                        a*=2;//9*2
                        System.out.println(a);
                        a/=2;//18/2
                        System.out.println(a);
                }
}
```

Control Structures

- ☐ Control flow statements are used to alter, redirect or to control the flow of program execution based on the application logic.
- ☐ Control flow statements are mainly classified as:
  1. Sequence structure
  2. Conditional control structure
  3. Unconditional control structure

1. Sequence statements

- ☐ Sequence control structure" refers to the line-by-line execution by which statements are executed sequentially, in the same order in which they appear in the program.
- ☐ Every program begins with the first statement of main(). Each statement in turn executed sequentially when the final statement of main() is executed.

2. Conditional control structure

- ☐ Sometimes the program needs to be executed depending upon a particular condition.
- ☐ Two types of conditional control structures:
  - a) Decision making statements or Selection statements or Branching statements
  - b) Looping statements or Iteration statements

a) Selection statements

- ❑ The Selection construct means the execution of statement(s) depending upon a condition-test.
- ☐ Two types of selection statements:
  - i) if statement

ii) switch statement

i) if statement

There are four different types of if statement in C++. These are:

- Simple if Statement
- if-else Statement
- Nested if Statement
- else-if Ladder

b) Looping Statements

- Three kinds of looping statements are:
  - for
  - while
  - do-whil

Example 1:-if

```
class IfExample
 {
        public static void main(String args[ ])
         {
           //defining an 'age' variable
          int age=20;
            //checking the age
                if(age>18)
                 {
                     System.out.print("Age is greater than 18");
                 }
          }
}
```

Example 2:-if-else

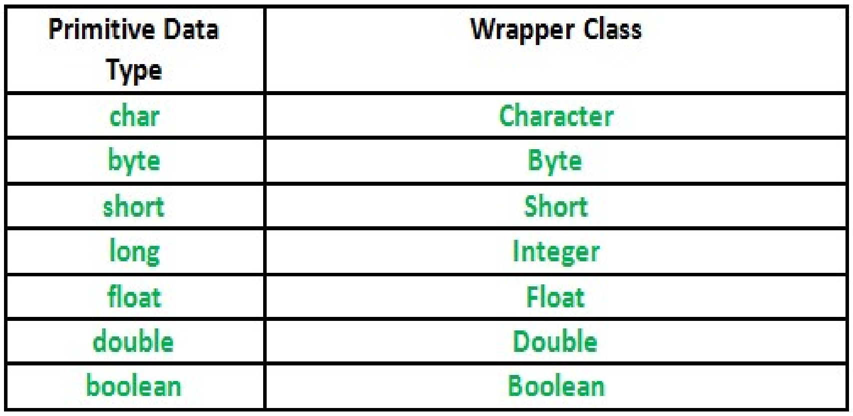
```
class IfElseExample
{
        public static void main(String args[ ])
         {
                    int number=13;
                        if(number%2==0)
                          {
                                 System.out.println("even number");
                          }
                    else
                    {
                              System.out.println("odd number");
                      }
          }
}
```

Example 3:-for

```
class ForExample
 {
          public static void main(String args[ ])
           {
                    //Code of Java for loop
                    for(int i=1;i<=10;i++)
                    {
                              System.out.println(i);
                     }
} }
```

Wrapper classes in Java

- ☐ In the **java**. lang package **java** provides a separate **class** for each of the primitive data types



Primitive Data Type	Wrapper Class
char	Character
byte	Byte
short	Short
long	Integer
float	Float
double	Double
boolean	Boolean

- ☐ Primitive data types can be converted into object type using the wrapper classes.
- ☐ The wrapper classes have a number of unique methods for handling primitive data types and objects.
  - Integer.parseInt(String) :-Converting numeric string to primitive integer

- Integer.toString(int) :- converting primitive integer to string

Example: 1

```
String s="24";

int a=Integer.parseInt(s);
System.out.print(a);
```

Example 2

```
        int num = 53;
String s=Integer.toString(num);
```

Command Line Arguments

- ✓ A program can accept an input at the time of execution.
- ✓ This is achieved in java program by using command line arguments.
- ✓ Command line arguments are parameters that are supplied to the program at the time of invoking it for execution.
- ✓ Java programs that can receive and use the arguments provided in the command line.
- ✓ The signature of the main( ) contains an argument which is of String type.
- ✓ Any arguments provided in the command line are passed to the array of String type which is declared with main( ) method.

Example:-factorial of a number

```
class FactorialExample
{
public static void main(String args[ ]) {
                int i,fact=1;
                   int number=Integer.parseInt(args[0]);
                        for(i=1;i<=number;i++)
                        {
                        fact=fact*i;
                        }
                        System.out.println("Factorial of "+number+" is: "+fact);
                }  }
```

This program is complied as,

```
javac FactorialExample.java
```

Run as

```
java FactorialExample 4
```

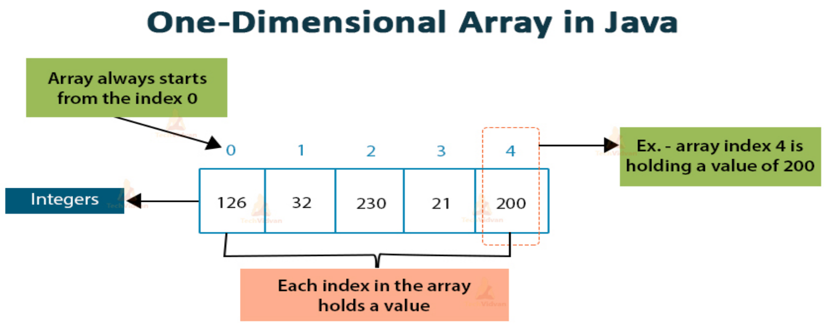
Output
Factorial of 4 is: 24

Arrays in Java

- An array is a collection of similar types of data.
- It is a container that holds values of one single type.
- For example, we can create an array that can hold 100 values of integer type.
- The data items put in the array are called elements and the first element in the array starts with index zero.
- Types of arrays
    - One dimensional Arrays
    - Multidimensional Arrays

One dimensional Arrays

- A one-dimensional array (or single dimension array) is a type of linear array.
- Accessing its elements involves a single subscript which can represent index.


One-Dimensional Array in Java
Array always starts from the index 0
0
1
2
3
4
Ex. - array index 4 is holding a value of 200
Integers
126
32
230
21
200
Each index in the array holds a value

Creating an Array

- Like any other variables, array must be declared and created in the computer memory before they are used.
- Creation of an array involves 3 steps:
    1. Declare the array
    2. Creating the memory locations
    3. Putting values into the memory locations

1. Declare the array

- Arrays in java can be declared in two forms

  datatype array_name[ ];

  Or

  datatype [ ] array_name;

    - We do not enter the size of the arrays in the declaration.

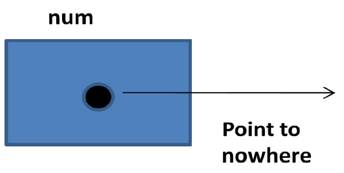
- For example,

  int num[ ];

  float avg[ ]; int [ ] marks;

❑ Representation of an array in memory,

int num[ ];



2. Creation of the array

- Array length is fixed at its creation time and cannot be changed
- Java allows us to create memory location for an array using new operator.
- ie, An array can create for a definite number of elements of a homogeneous type.
- The general form is:

  array_name = new datatype[size];

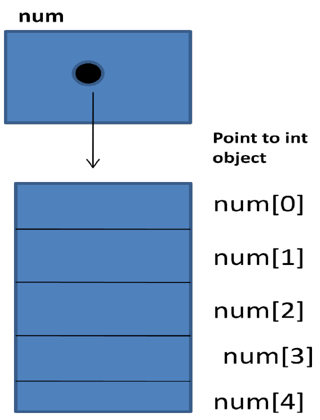
- **Example**

  num = new int[10];

  avg = new float[10];

❑ **Creation of an array in memory**

num = new int[5];



- It is also possible to combine the declaration and creation into one.
- The general form is

  datatype arrayname[ ] = new datatype [size];

- **Example**

  int num [ ] = new int[10];

  float avg[ ] = new float[10];

3. Initialization of array

- The final step is to put values into the array created.
- This process is known as initialization.
- This can be done in two ways

a) Using the array subscript as,

array_name[subscript]=value;

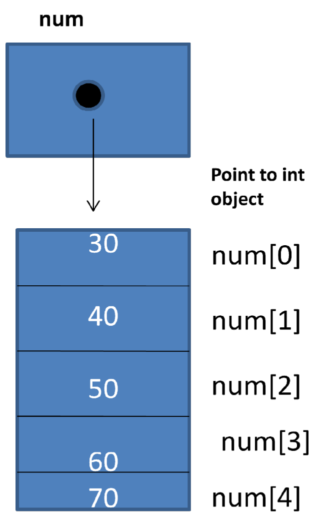
- For example,

```
num[0]=30;
num[1]=40;
num[2]=50;
num[3]=60;
num[4]=70;
```

- This is also possible as,

```
int value=30;
for(int i=0;i<5;i++)
{
        num[i]=value;
        value+=10;
}
```

- This can be represented as



b) initialize with list of values

- we can also initialize arrays automatically in the same way as the ordinary variables when they are declared.

- The general form is,
  datatype array_name[ ]={list of values};
  - The list of values are separated by commas and surrounded by curly brackets.
- No need for given the size of the array.
- The compiler allocates enough space for all the elements specified in the list.

- For example,
  int a[ ]={35,45,55,65};
- Here automatically the array size is set as 4.

Array length

- In java all arrays store the allocated size in a variable named **length**.
- For example,
  int a[ ]={35,45,55,65};
  int size=a.length;
  System.out.print(size);

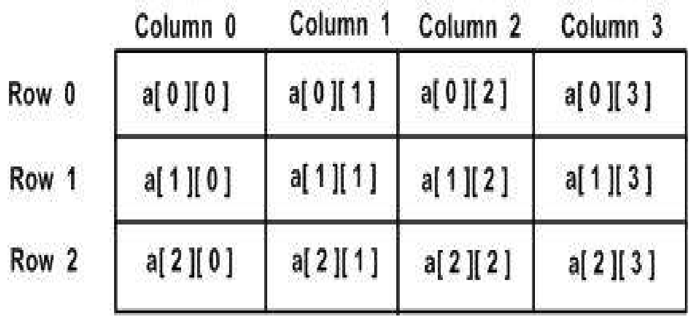
/*Sort an integer array using command line arguments*/

```
class sort
{
        public static void main(String args[])
        {
        int num[ ]=new int[10];
        int i;
        int len=args.length;
        System.out.println("Array length="+len);
        System.out.println("Given list");
        for(i=0;i<len;i++)
                {
                num[i]=Integer.parseInt(args[i]);
                System.out.println(num[i]);
                }
        for(i=0;i<len;i++)
        {
        for(int j=0;j<len;j++)
        {
        if(num[i]<num[j])
        {
```

```
                    int temp=num[i];
                    num[i]=num[j];
                    num[j]=temp;
            }
            }
            }
System.out.println("Sorted list list");
for(i=0;i<len;i++)
{
System.out.println(num[i]);
}
}
}
```

Two dimensional arrays

- In the two-dimensional array, each element associated with two indexes. The most common use of two dimensional array is matrix manipulations. Two dimensional arrays are stored in memory a shown in the figure,



	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

- This figure represents an array a[3][4].
- That is it is 3X4 array named as a.
- Each dimension of the array is indexed from zero to its maximum.
- The first index represents the row and the second index represents the column.
- Creation of two dimensional arrays also involves 3 steps.
    - Declare the array
    - Creating the memory locations
    - Putting values into the memory locations

1. Declare the array

- The general form is,
  datatype arr_name[ ][ ];

For example,

int a[ ][ ];

2. Creating the memory location

- The general form is,

arr_name=new datatype[row_size][col_size];

- For example,
  a=new int[3][4];
- It can be also use as,
  int a[ ][ ]=new int[3][4];

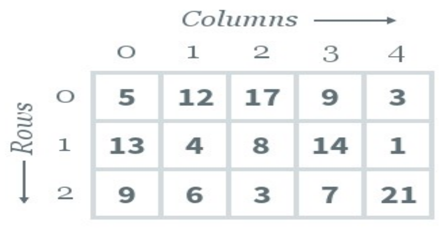
3. Initialization of the array

- This can be done in two ways.

a) The general form is,

datatype arr_name[row_size][col_size]={list of values};

- For example,
  int mat[3][5]={5,12,17,9,3,13,4,8,14,1,9,6,3,7,21};
  - Here create a 3X5 matrix.
  - This elements are stored in the memory as



2D Array of size 3 x 5

b) General form is,

datatype arr_name[][]={{list of first row},{list of second row},.....};

For example,

int mat[ ][ ]={{5,12,17,9,3},{13,4,8,14,1},{9,6,3,7,21}};

Commas are required after each curly brackets that closes off a row.

Program for addition of two matrices

```
class MatrixAddition
{
        public static void main(String args[])
        {
            //creating two matrices
        int a[][]={{1,3,4},{2,4,3},{3,4,5}};
        int b[][]={{1,3,4},{2,4,3},{1,2,4}};
        //creating another matrix to store the sum of two matrices
        int c[][]=new int[3][3];  //3 rows and 3 columns
        //adding and printing addition of 2 matrices
        for(int i=0;i<3;i++)
        {
            for(int j=0;j<3;j++)
            {
                c[i][j]=a[i][j]+b[i][j];
             }
        }
        for(int i=0;i<3;i++)
        {
            for(int j=0;j<3;j++)
            {
                System.out.print(c[i][j]+" ");

            }
            System.out.println();//new line
        }
        }
}
```

$$a = \begin{Bmatrix} 1 & 3 & 4 \\ 2 & 4 & 3 \\ 3 & 4 & 5 \end{Bmatrix} \qquad b= \begin{Bmatrix} 1 & 3 & 4 \\ 2 & 4 & 3 \\ 1 & 2 & 4 \end{Bmatrix}$$

$$\begin{matrix} a \\ + \\ b \end{matrix} \begin{Bmatrix} 1+1 & 3+3 & 4+4 \\ 2+2 & 4+4 & 3+3 \\ 3+1 & 4+2 & 5+4 \end{Bmatrix}$$

$$C = \begin{Bmatrix} 2 & 6 & 8 \\ 4 & 8 & 6 \\ 4 & 6 & 9 \end{Bmatrix}$$

String class in Java

- ☐ Generally, String is a sequence of characters.
- ☐ But in Java, string is an object that represents a sequence of characters.
- ☐ The java.lang.String class is used to create a string object.
- ☐ Two classes are used to handle strings in java
  - String
  - StringBuffer

String class

- ☐ There are two ways to create String object:
  - By new keyword
  - By string literal

1. Creating String object using new operator

a) A character array can be converted as a String object using the constructor.

- The syntax is

```
String s=new String(Char c[ ]);
```

For eg:

```
char ch[ ]={'s','t','r','i','n','g','s'};
//converting char array to string
String s1=new String(ch);
System.out.println(s1);
```

Output

```
strings
```

b) Creating a string object from a byte array

- The syntax is,

```
String(byte a[ ]);
```

For eg:

```
byte asci[ ]={65,66,67,68,69,70};
String s=new String(asci);
System.out.println(s);
```

Output

```
ABCDEF
```

2. Creating String as literal

- Create a constant string
- For eg;

  String s="Welcome"

Methods in String class

- The java.lang.String class provides many useful methods to perform operations on String class.

Method	Description
char charAt(int index)	returns char value for the particular index
int length( )	returns string length
String substring(int startindex)	returns substring for given beginning index
String substring(int beginIndex, int endIndex)	returns substring for given begin index and end index.
String concat(String str)	Concatenates specified string to the end of this string.
String toLowerCase()	Converts all the characters in the String to lower case
String toUpperCase()	Converts all the characters in the String to upper case.
String replace (char oldchar, char nwchar)	Returns new string by replacing all occurrences of *oldchar* with *newchar.*
String replace(String old, String new):	replaces all occurrences of old string to new string

int indexOf (String s)	Returns the index within the string of the first occurrence of the specified string.
int indexOf (String s, int i)	Returns the index within the string of the first occurrence of the specified string, starting at the specified index
Int lastIndexOf(String s)	Returns the index within the string of the last occurrence of the specified string.
boolean equals(String str)	Compares two strings. Returns true if the strings are equal, and false if not
boolean equalsIgnoreCase(String str)	Compares two strings, ignoring case considerations
int compareTo(String string2)	☐ Compares two string lexicographically. ▣ The method returns 0 if the string is equal to the other string. ▣ A value less than 0 is returned if the string is less than the other string (less characters) and a value greater than 0 if the string is greater than the other string (more characters).
int compareToIgnoreCase()	to compare two strings lexicographyically, ignoring lower case and upper case differences

Example 1

```
class StringEg1
{
        public static void main(String args[])
        {
        String s="Welcome to Java";
```

```
            char ch=s.charAt(4);
             System.out.println(ch);
            int len=s.length( );
            System.out.println("string length is: "+len);
            System.out.println(s.substring(2));
            System.out.println(s.substring(2,4));
            }
}
```

Example 2

```
class stringEg2
{
        public static void main(String args[ ])
        {
        String s1="my name is JAVA";
        String s2="  It is opps";
        //replaces all occurrences of 'a' to 'e'
        String newString=s1.replace('a','e');
        System.out.println(newString);
        //replaces all occurrences of "is" to "was"
        String new=s1.replace("is","was");
        System.out.println(new);
        String new2=s1.concat(s2);
        System.out.println(new2);
        String lower=s1.toLowerCase();
        System.out.println(lower);
        String  upper=s1.toUpperCase();
        System.out.println(upper);
        }
}
```

- **Example 3**

```
String s = "Learn Share Learn";
int output = s.indexOf("Share"); // returns 6
int output = s.indexOf("ea",3);// returns 13
 int output = s.lastIndexOf("a"); // returns 14
```

- **Example 4**

```
String myStr1 = "Hello";
String myStr2 = "Hello";
```

```
                String myStr3 = "Another String";
                System.out.println(myStr1.equals(myStr2));
                // Returns true because they are equal
                System.out.println(myStr1.equals(myStr3)); // false
                String myStr4 = "HELLO“;
                System.out.println(myStr1.equalsIgnoreCase(myStr4)); // true
                System.out.println(myStr1.equals(myStr4)); //false
```

☐ **Example 5 for sort a string array**

```
class SortStringArrayExample1
{
    public static void main(String args[ ])
    {
                //defining an array of type String
        String countries [ ]= {"Zimbabwe", "South-Africa", "India", "America”, "France“,
                            "Italy", "Germany"};
         int size = countries.length;
                for(int i = 0; i<size-1; i++)
                {
                     for (int j = i+1; j<size; j++)
                      {
                      //compares each elements of the array to all the remaining elements
                       if(countries[i].compareTo(countries[j])>0)
                         {
                                  //swapping array elements
                              String temp = countries[i];
                               countries[i] = countries[j];
                            countries[j] = temp;
                         }
                     }
                }
//prints the sorted array in ascending order
                System.out.println(“Sorted list”);
                for(int i=0; i<size; i++)
                {
                        System.out.println(countries[i]);
                }
    }
}
```

StringBuffer class

- **StringBuffer** is a peer class of **String** that provides much of the functionality of strings.
- String represents fixed-length, unchangeable character sequences while StringBuffer represents grow able and writable character sequences.
- **StringBuffer** may have characters and substrings inserted in the middle or appended to the end.
- It will automatically grow to make room for such additions and often has more characters reallocated than are actually needed, to allow room for growth.

StringBuffer Constructors

- **StringBuffer( )**
  - StringBuffer s=**new** StringBuffer();
  - It reserves room for 16 characters without reallocation.

- **StringBuffer(int size)**
  - StringBuffer s=**new** StringBuffer(20);
  - It accepts an integer argument that explicitly sets the size of the buffer.

- **StringBuffer(String str):**
  - StringBuffer s=**new** StringBuffer(“Welcome");
  - It accepts a **String** argument that sets the initial contents of the StringBuffer object and reserves room for 16 more characters without reallocation.

Methods

- **int length( ) and int capacity( ):**
  - The length of a StringBuffer can be found by the length( ) method, while the total allocated capacity can be found by the capacity( ) method.
  - Ex.

```
StringBuffer s = new StringBuffer(“Welcome");
 int p = s.length();
int q = s.capacity();
System.out.println("Length of string=" + p); //7
System.out.println("Capacity of string=" + q); //7+16=23
```

- ☐ **StringBuffer insert(int index, String str)**
  - It is used to insert text at the specified index position.
- ☐ **StringBuffer reverse( ):**
  - It can reverse the characters within a StringBuffer object.
- ☐ **StringBuffer delete(int startIndex, int endIndex)**
  - It can delete characters within a StringBuffer.
  - Here, start Index specifies the index of the first character to remove, and end Index specifies an index one past the last character to remove.
  - Thus, the substring deleted runs from start Index to endIndex–1. The resulting StringBuffer object is returned.
- ☐ **StringBuffer deleteCharAt(int loc)**
  - This method deletes the character at the index specified by *loc.*
  - It returns the resulting StringBuffer object.
- ☐ **StringBuffer replace(int startIndex, int endIndex, String str)**
  - It can replace one set of characters with another set inside a StringBuffer object by calling replace( ).
  - The substring being replaced is specified by the indexes start Index and endIndex.
  - Thus, the substring at start Index through endIndex–1 is replaced.
  - The replacement string is passed in str.The resulting StringBuffer object is returned
- ☐ **String toString( )**
  - This method returns a string representing the data in this sequence

- ☐ **a) StringBuffer append(String str)**

  b) StringBuffer append(int n)
  - This method will concatenate the string representation of any type of data to the end of the StringBuffer object.

- ☐ **Example 1**

```
StringBuffer str = new StringBuffer("test"); str.append(123);
System.out.println(str);
```

Output

```
test123
```

- ❑ **Example 2**

```
StringBuffer str = new StringBuffer("Hello"); str.reverse();
System.out.println(str);
```

Output

```
olleH
```

- ☐ **Example 3**

```
StringBuffer s = new StringBuffer(“Welcome");
 s.delete(0, 4);
System.out.println(s);
s.deleteCharAt(6);
System.out.println(s);
```

Output

```
me
m
```

- ☐ **Example 4**

```
StringBuffer str = new StringBuffer("Hello World");
str.replace( 6, 11, "java");
System.out.println(str);
```

Output

```
Hello java
```

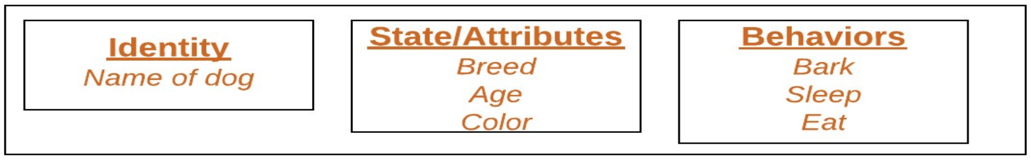
Classes and objects in Java

Class

- Classes and Objects are basic concepts of Object Oriented Programming which revolve around the real life entities.
- Anything we wish to represent in a Java program must be encapsulated in a class.
- A class is a user defined blueprint or prototype from which objects are created.
- It represents the set of properties or methods that are common to all objects of one type.

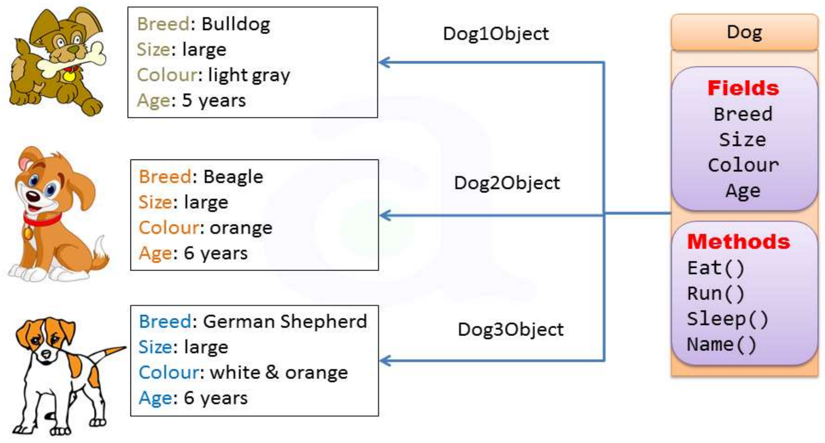
Object

- Object Definitions:
    - An object is a real-world entity.
    - An object is a runtime entity.
    - The object is an entity which has state and behavior.
    - The object is an instance of a class.
- An object has three characteristics:
    - **State** : It is represented by attributes of an object. It also reflects the properties of an object.
    - **Behavior** : It is represented by methods of an object. It also reflects the response of an object with other objects.
    - **Identity** : It gives a unique name to an object and enables one object to interact with other objects.
- Example of an object : dog



- When an object of a class is created, the class is said to be **instantiated**.

- All the instances share the attributes and the behavior of the class.
- But the values of those attributes, i.e. the state are unique for each object.
- A single class may have any number of instances.



Defining a class

- A class is a user-defined data type with a template that serves to define its properties.
- Basic form of class definitions is

```
Class class_name
  {
           variable declaration;
          method deceleration
  }
```

Field declaration

Data is encapsulated in a class by placing data fields inside the body of class definition. These variables are called instance variables because they are created whenever an object of the class is instantiated. We can declare the instance variables exactly the same way as we declare local variables.

Eg.1, **class** Student

```
{
        int id;
        String name;
    }
```

Eg.2, class Rectangle

```
    {
        int length;
        int width;
    }
```

Method declaration

- The functions declared inside the class are called methods.
- Methods are declared inside the class immediately after the declaration of instance variables.
- The general form of method declaration is:

```
type meth_nam(prameter-list)
{
    body of method;
}
```

- Method declaration have four basic parts:
    - The name of the method:- meth_name()
    - The type of the value the method returns:- It could be void, if the method does not return any value
    - A list of parameters
    - The body of the method

Eg1:

```
class Student
{
    int rollno;
    String name;
    void insert(int r, String n)
    {
```

```
                rollno=r;
                name=n;
          }
        void display()
        {
                System.out.println(rollno+" "+name);}
        }
}
```

- **Eg2,**

```
class Rectangle
{
                int length, width;
        void insert(int l, int w)
        {
                length=l;
                width=w;
         }
        int rectArea()
        {
                int area=length*width;
                return(area);
        }
}
```

Creating Objects

- An object in java is essentially a block of memory that contains a space to store all the instance variables.
- Creating an object is also referred to as instantiating an object.
- When we create a class, we are creating a new data type.
- We can use this type to declare objects of that type.
- However, obtaining objects of a class is a two-step process.

1. **Declare a variable of the class type.**

- This variable does not define an object.
- Instead, it is simply a variable that can *refer to an object.*

- *The syntax is,*

  class_name obj_name;

  - *For example,*

    Student stud;

    Rectangle myrect;

2. **Instantiate the object**

- We must acquire an actual, physical copy of the object and assign it to that variable
- We can do this using the new operator.
- The new operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it.
- This reference is, more or less, the address in memory of the object allocated by new.
- The syntax is,

  obj_name = new class_name( );

  - For example ,

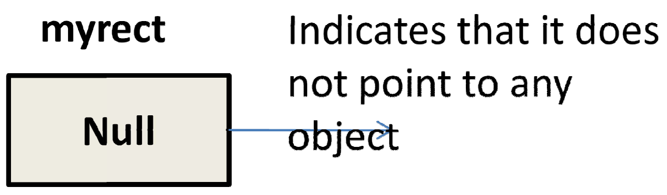
    sud =new Student ( );

    myrect= new Rectangle( );

- The statement

  Rectangle myrect;   // declare the object

- This line declares **myrect as a reference to an object of type rectangle.**
- **After this line** executes, myrect contains the value **null, which indicates that it does not yet point to an actual** object.

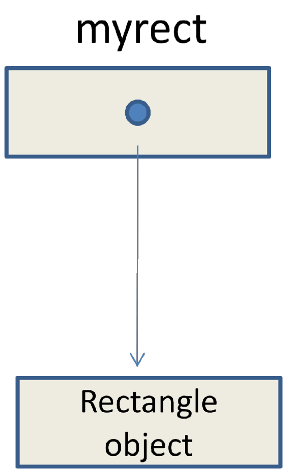


- Any attempt to use myrect at this point will result in a compile-time error.

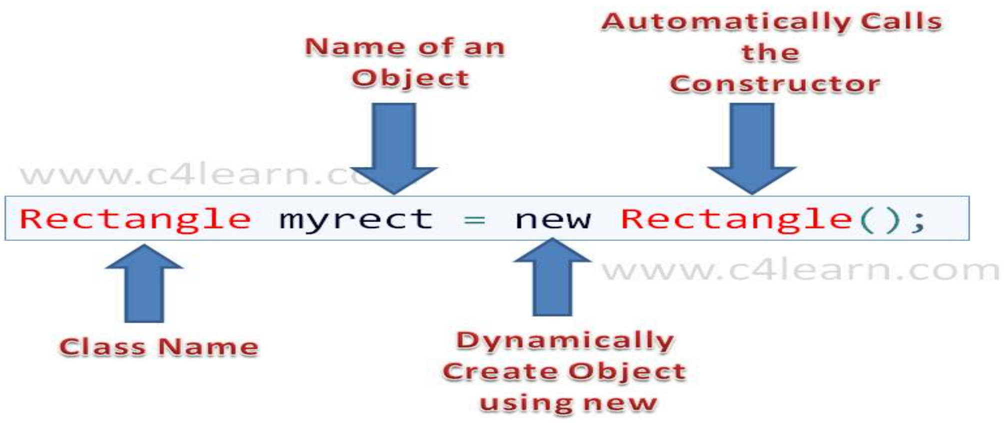
- The next line,

  myrect = new Rectangle( );

- This statement allocates an actual object and assigns a reference to it to **myrect.**
- **After the second line** executes; we can use myrect **as if it were a demo object.**
- **But in reality, myrect simply holds the** memory address of the actual demo object.
- The effect of this line of code is depicted in Figure.



- Both statements can be combined into one as shown below.


Name of an Object
Automatically Calls the Constructor
www.c4learn.com
Rectangle myrect = new Rectangle();
www.c4learn.com
Class Name
Dynamically Create Object using new

Accessing class members

- All the variables must be assigned values before they are used.
- Since we are outside the class, we cannot access the instance variables and methods directly.

- To do this, we must use the concerned object and the dot(.) operator.
- The instance variables are accessed as,

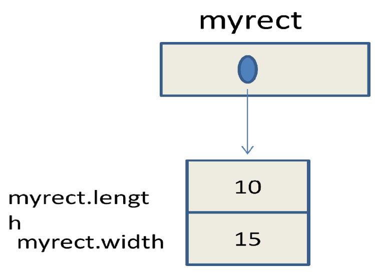
  obj_name.variable=value;

  - Here obj_name is the object name, variable is the name of the instance variable inside the object that we wish to access.
  - For example,

myrect.length=10;

myrect.width=15;

Now the object myrect is



- **Example program**

```
class Rectangle
{
                int length, width;
}
class RectArea
{
        public static void main(String arg[ ])
        {
                Rectangle myrect=new Rectangle( );
                myrect.length=10;
                myrect. Width=15;
                int area=myrect.length*myrect.width;
                System.out.printl("Area="+area);
        }
}
```

- The methods are also accessing with the help of concerned object and dot (.) operator.
- The general form is,

```
                    obj_name. meth_name( );
```

- The sample program is,

```
class Rectangle
{
          int length, width;
          void rectArea()
          {
            int area=length*width;
            System.out.printnnl("Area="+area);
          }
}

class RectArea
{
   public static void main(String arg[ ])
   {
  Rectangle myrect=new Rectangle( );
        myrect.length=10;
        myrect. Width=15;
        myrect.rectArea( );
        }
}
```

- There are two convenient ways for accessing and assigning values to the instance variables.
    1. Use a method that is declared inside that class
    2. Use the constructor

1. Initializing instance variables through methods

- ☐ Here the values are assigned to instance variables through the methods.
- ☐ For example,

```
class Rectangle
{
                int length, width;
        void insert(int l, int w)
        {
                length=l;
```

```
            width=w;
        }
        int rectArea()
        {
            int area=length*width;
            return(area);
        }
}

class RectArea
{
    public static void main(String arg[ ])
    {
        Rectangle rect=new Rectangle( );
        rect1.insert(20,30);
        int a=rect1.rectArea( );
        System.out.println("Area="+a);
    }
}
```

2. Constructors in Java

- Constructor is a special type of method which is used to initialize the object.
- Constructor name must be the same as its class name
- A Constructor must have no explicit return type, not even void.
- A Java constructor cannot be abstract, static, or final.
- It is called constructor because it constructs the values at the time of object creation.
- At the time of calling constructor, memory for the object is allocated.
- Every time an object is created using the new keyword, at least one constructor is called.
- It calls a default constructor if there is no constructor available in the class.
- In such case, Java compiler provides a default constructor by default.

☐ There are two types of constructors in Java:
  - ✓ no-arg constructor or default constructor
  - ✓ parameterized constructor.

Default constructor

- ✓ A constructor is called "Default Constructor" when it doesn't have any parameter.
- ✓ Syntax of default constructor:

```
class_name( )
{
        ------------
}
```

- ✓ These will be invoked at the time of object creation with no initial values.
- ✓ The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.
- ✓ If there is no constructor in a class, compiler automatically creates a default constructor.

Example for default constructor

```
class Student
{
        int id;
        String name;
        Student( )
        { }
        //method to display the value of id and name
        void display( )
        {
        System.out.println(id+" "+name);
        }
}

class stud
{
        public static void main(String args[])
        {
                //creating objects
                Student s1=new Student();
                Student s2=new Student();
                //displaying values of the object
                s1.display();
                s2.display();
        }
}
```

Output

```
0  null
0  null
```

Parameterized Constructor

- ☐ A constructor which has a specific number of parameters is called a parameterized constructor.
- ☐ The parameterized constructor is used to provide different values to distinct objects.

Example for parameterized constructor

```
class Student
{
        int id;
        String name;
        Student(int n, String s)
        {
                id=n;
                name=s;
         }
        //method to display the value of id and name
        void display( )
        {
        System.out.println(id+" "+name);
        }
}

class stud
{
        public static void main(String args[ ])
        {
                //creating objects
                Student s1=new Student(12,"Anu");
                Student s2=new Student(13,"Ram");
                //displaying values of the object
                s1.display();
                s2.display();
        }
}
```

Method Overloading

- Method Overloading is a feature of java that allows a class to have more than one method having the same name.
- Java can distinguish between overloaded methods with different method signatures.
- That is in an overloaded method, the argument lists of the methods must differ in either of these:

1. Number of parameters.

- For eg:
  void add(int x, int y)
  void add(int x, int y, int z)

2. Data type of parameters.

- For eg:
  void add(int x, int y)
  void add(int a, float b)

3. Sequence of Data type of parameters.

For eg:
    void add(int a, float b)
    void add(float a, int b)

- In java, method overloading is not possible by changing the return type of the method only.
- For example if two methods have same name, same parameters and have different return type, then this is not a valid method overloading example.
- This will throw compilation error.

- That is
  - int add(int, int)
  - float add(int, int)
  - The above two statements will throw compilation error.

Example 1:- Overloading – Different Number of parameters in argument list

This example shows how method overloading is done by having different number of parameters

```
class Adder
{
        int add(int a,int b)
        {
                return a+b;
        }
         int add(int a,int b,int c)
        {
                return a+b+c;
        }
}
class TestOverloading1
{
        public static void main(String args[ ])
        {
          Adder obj1=new Adder( );
         int s1=obj1. add(10,20);
         int s2=obj1.add(10,20,30);
         System.out.println(‘Sum 1=“+s1);
         System.out.println(“Sum 2=“+s2);
         }
}
```

Output

```
Sum1 =30
Sum 2=60
```

Example 2:- Overloading –Difference in data type of parameters

In this example, we have created two methods that differs in data type. The first add method receives two integer arguments and second add method receives two double arguments.

```
class Adder2
{
         int add(int a, int b)
        {
                return a+b;
        }
```

```
          double add(double a, double b)
          {
          return a+b;
          }
}
class TestOverloading2
{
          public static void main(String[] args)
          {
            Adder2 obj2=new Adder2( );
            int s1=obj2. add(10,20);
            int s2=obj2.add(12.3,12.6);
           System.out.println(‘Sum1=“+s1);
           System.out.println(“Sum2=“+s2);
           }
}
```

<u>Output</u>

```
Sum1 =30
Sum 2=24.9
```

Example 3:- Overloading- change the sequence of data type of arguments

```
class student
{
           void Identity(String name,  int id)
          {
                    System.out.println(“Name 1:"+ name +" "+"Id :"+ id);
          }
           void Identity(int id, String n)
          {
                     System.out.println(“Name2:"+ n +" "+"Id :"+ id);
           }
}
class overloading3
{
public static void main (String args[ ])
 {
    student stu = new student();
      stu.Identity("Mohit", 1);
      stu.Identity(2, “Ram");
```

```
   }
}
```

Constructor Overloading

- In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.
- Constructor overloading in Java is a technique of having more than one constructor with different parameter lists.
- They are arranged in a way that each constructor performs a different task.
- They are differentiated by the compiler by the number of parameters in the list and their types.

Example of Constructor Overloading

```
class StudentData
{
        int id;
         String name;
         int age;
        // creating default constructor
        StudentData()
         {
                id = 100;
                name = "New Student";
                age = 18;
        }
         //creating two arg constructor
         StudentData(int i,String n)
        {
           id = i;
           name = n;
          }

 //creating three arg constructor

        StudentData(int i,String n,int a)

        {
                id = i;
                name = n;
                age=a;
        }
        void display()
        {
        System.out.println(id+" "+name+" "+age);
        }
}
class Student
{
        public static void main(String args[ ])
        {
        StudentData s1 = new StudentData( );
        StudentData s2 = new StudentData(111,"Karan");
        StudentData s3 = new StudentData(222,"Aryan",25);
                s1.display();
                s2.display();
                s3.display( );
         }
}
```

Static Members

- We can declare a class basically contains two sections:
    - Declare instance variables
    - Declare the instance methods
- These are called instance members of a class.
- Because every time the class is instantiated, a new copy of each of them is created.
- They are accessed using the objects.
- If we want to define a member that is common to all objects and accessed without using a particular object.
- That is a member belongs to the class as a whole rather than the objects created from the class.
- Such members are called static members.
- We can create static variables and static methods using the keyword static.
- For eg.

```
                static int cout;

                static int max(int x, int y);
```

Since these members are associated with the class itself rather than individual objects, the static variables and static methods are often referred to as class variables and class methods.

Static Variables or class variables

- Static variables are also called **class variables** because they can be accessed using class name.
- Static variables are used when we want to have a variable common to all instances of a class.
- Static variables occupy single location in the memory.
- These are accessible in both static and non-static methods, even non-static methods can change their values.
- Commonly the static variables are using in the case of we want to keep record about the number of running instances (objects) of a class.
- **Example for static variable**

```
class Student
 {
        static int count=0; //static variable
        Student( )
         {//increment static variable
                count++;
        }
        void showCount( )
        {
                System.out.println("Number Of students : "+count);
        }
 }
class StaticDemo
{
public static void main(String  args[ ])
{
Student s1=new Student();
                s1.showCount();
                Student s2=new Student();
```

```
            s2.showCount();
            Student s3=new Student();
            Student s4=new Student();
            s3.showCount();
            s4.showCount();
            Student.count=20;
            s4.showCount();
        }
}
```

Output

```
Number Of students : 1
Number Of students : 2
Number Of students : 4
Number Of students : 4
Number Of students: 20
```

Static Methods/Class Methods

- When a method is declared with *static* keyword, it is known as static method.
- The most common example of a static method is *main()* method.
- Like static variables, static methods can be called without using the objects.

Example program for static method

```
class mathop
{
        static float mul(float x, float y)
        {
                return  x*y;
        }
        static float div(float a, float b)
        {
                return x/y;
        }
}
class staticmeth
{
        public static void main(String arg[])
        {
                float m=mathop.mul(14.5,10.2);
                float d=mathop.div(m,2.0);
                System.out.println("Product="+m);
                System.out.println("divide="+d);
        }
}
```

Inheritance

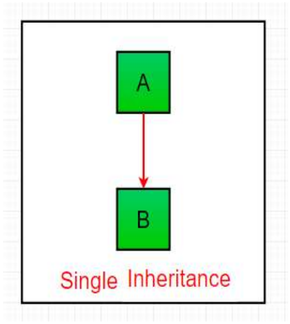
- Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another.
- The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class).
- extends Keyword
    - **extends** is the keyword used to inherit the properties of a class. Following is the syntax of extends keyword.
- **The Syntax is**

```
class base_class
{
 ..... .....
}
class sub_class extends base_class
{
 ..... .....
}
```

- The **extends keyword** indicates that we are making a new class that derives from an existing class.
- The meaning of "extends" is to increase the functionality.

Types of Inheritance in Java

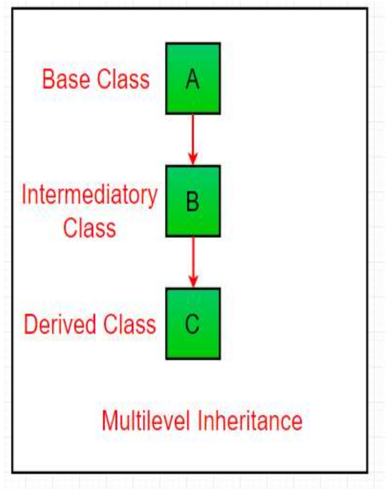
1. Single Inheritance : In single inheritance, subclasses inherit the features of one super class. In image below, the class A serves as a base class for the derived class B.



Example program for single level inheritance

```
class Rectdemo
{
int le,be;
void getval(int a,int b)
{
        le=a;
        be=b;
}
int rectArea( )
{
        return le*be;
}
}
class Triangle extends Rectdemo
{
        int b,h;
        float t;
void  getdata(int q,int r)
{
        b=r;
        h=q;
}
float triArea( )
{
        t=(float)le/2*b*h;
        return (t);
}
}
class single_inher
{
public static void main(String args[])
{
        Triangle Tr=new Triangle();
        Tr.getval(40,8);
        Tr.getdata(10,6);
        System.out.println("Area of Rectangle:"+  Tr.rectArea());
        System.out.println("Area of Triangle:"+Tr.triArea());
}
}
```

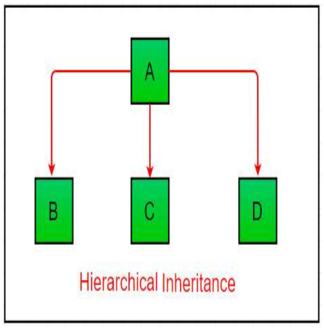
2. Multilevel Inheritance : In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class. In below image, the class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C.



Syntax is:

```
class A
{
  ..............
}
class B extends A
{
   .....................
}
class c extends B
{
  ......................................
}
```

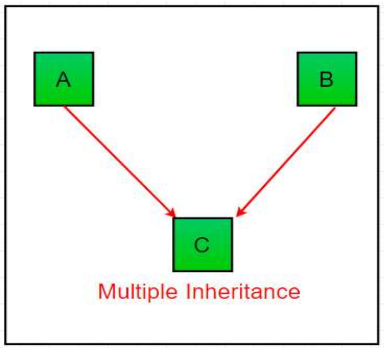
3. Hierarchical Inheritance : In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one sub class.In below image, the class A serves as a base class for the derived class B,C and D.



Syntax is:

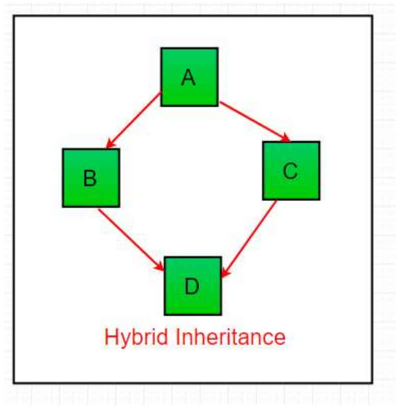
```
class A
{
  .............
}
class B extends A
{
   .....................
}
class c extends A
{
  ........................
}
class D extends A
{
  ............
}
```

4. **Multiple Inheritance :** In Multiple inheritance ,one class can have more than one superclass and inherit features from all parent classes. Java does **not** support multiple inheritance with classes. In java, we can achieve multiple inheritance only through Interfaces. In image below, Class C is derived from interface A and B.



class C extends A,B – this statement is invalid

4. **Hybrid Inheritance :** It is a mix of two or more of the above types of inheritance. Since java doesn't support multiple inheritance with classes, the hybrid inheritance is also not possible with classes. In java, we can achieve hybrid inheritance only through Interfaces.



Example java program for multilevel inheritance- calculate the students total mark

```
class stud_details
{
    private int sno;
    private String sname;
    void setstud(int no,String name)
        {
                sno = no;
               sname = name;
         }
        void putstud()
        {
              System.out.println("Student No : " + sno);
              System.out.println("Student Name : " + sname);
        }
}

class marks extends stud_details
{
     protected int mark1,mark2;
     public void setmarks(int m1,int m2)
     {
          mark1 = m1;
          mark2 = m2;
     }
     public void putmarks()
     {
       System.out.println("Mark1 : " + mark1);
        System.out.println("Mark2 : " + mark2);
     }
}
```

```
class finaltot extends marks
{
        private int total;
         void calc()
        {
                total = mark1 + mark2;
        }
        public void puttotal()
        {
                 System.out.println("Total : " + total );
         }
}

class student
{
public static void main(String args[])
      {
            finaltot f = new finaltot();
            f.setstud(100,"Nithya");
            f.setmarks(78,89);
            f.calc();
            f.putstud();
            f.putmarks();
            f.puttotal();
      }
}
```

<u>Super keyword in Java</u>

- The **super** keyword in Java is a reference variable which is used to refer immediate parent class object.
- Whenever we create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

The use of super keyword

1) To access the data members of parent class when both parent and child class have member with same name.

Example 1 for super keyword

```
class Superclass
 {
        int num = 100;
 }
 class Subclass extends Superclass
 {
        int num = 110;
        void printNumber()
        {
                System.out.println("First number="+super.num);
                System.out.println("second number="+num);
        }
}
class example_super
{
public static void main(String args[ ])
{
        Subclass obj= new Subclass( );
        obj.printNumber();
}
}
```

Output

```
First number=100
Second number=110
```

2) Invoke the parent class constructor.

- The keyword "super" is used to invoke the super class's constructor from within subclass's constructor.
- The statement super must be the first line of child class constructor. Calling a parent class constructor's name in the child class causes syntax error.

- **Example 2:- calling a default constructor using super**

```
class Person
{
         Person()
        {
        System.out.println("Person class Constructor");
         }
}
 /* subclass Student extending the Person class */
class Student extends Person
{
        Student()
        {
       // invoke or call parent class constructor
              super();
            System.out.println("Student class Constructor");
         }
}
class Test
{
   public static void main(String args[ ])
   {
      Student s = new Student();
   }
}
```

- **Example 3:- calling a parameterized constructor using super**

```
class Person
{
        int id;
        String name;
        Person(int i,String n)
        {
                id=i;
                name=n;
        }
}
class Emp extends Person
{
        float salary;
        Emp(int i,String n,float s)
        {
        super(i,n);
        salary=s;
        }
void display()
```

```
 {
System.out.println(id+" "+name+" "+salary);
}
}
class Test
{
public static void main(String args[ ])
{
        Emp e1=new Emp(1,"ankit",45000f);
        e1.display();
}
}
```

Method overriding

- Declaring a method in **sub class** which is already present in **parent class** is known as method overriding.
- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- In overriding, method of both class **must** have **same name** and equal number of **parameters with similar datatype**.
- They have same method signature in sub classes with different method body.
- Constructors cannot be overridden.
- Example program1

```
// Base Class
class A
{
        int num1,num2;
        A( int a, int b)
        {
                num1=a;
                num2=b;
        }
        void show()
        {
        System.out,println("Number1="+num1);
        System.out,println("Number2="+num2);
        }
}
```

```
//Derived class
class B extends A
{
        int num3;
        B(int a, int b, int c)
        {
                super(a,b);
                num3=c;
        }
        void show( )
        {
                System.out,println("Number3="+num3);
        }
}

class test_override1
{
        public static void main(String args[ ])
        {
                B ob=new B(10,20,30);
                ob.show( );
        }
}
```

Output

Number3=30

- In method override the version of a method that is executed will be determined by the object that is used to invoke it.
- If an object of a parent class is used to invoke the method, then the version in the parent class will be executed, but if an object of the subclass is used to invoke the method, then the version in the child class will be executed.
- In other words, *it is the type of the object being referred to* (not the type of the reference variable) that determines which version of an overridden method will be executed.
- We can call parent class method in overriding method using super keyword.

```
// Base Class
class A
```

```
{
        int num1,num2;
        A( int a, int b)
        {
                num1=a;
                num2=b;
        }
        void show()
        {
        System.out,println("Number1="+num1);
        System.out,println("Number2="+num2);
        }
}
//Derived class
class B extends A
{
        int num3;
        B(int a, int b, int c)
        {
                super(a,b);
                num3=c;
        }
        void show( )
        {
                super.show( );
        System.out,println("Number3="+num3);
        }
}

class test_override1
{
        public static void main(String args[ ])
        {
                B ob=new B(10,20,30);
                ob.show( );
        }
}
```

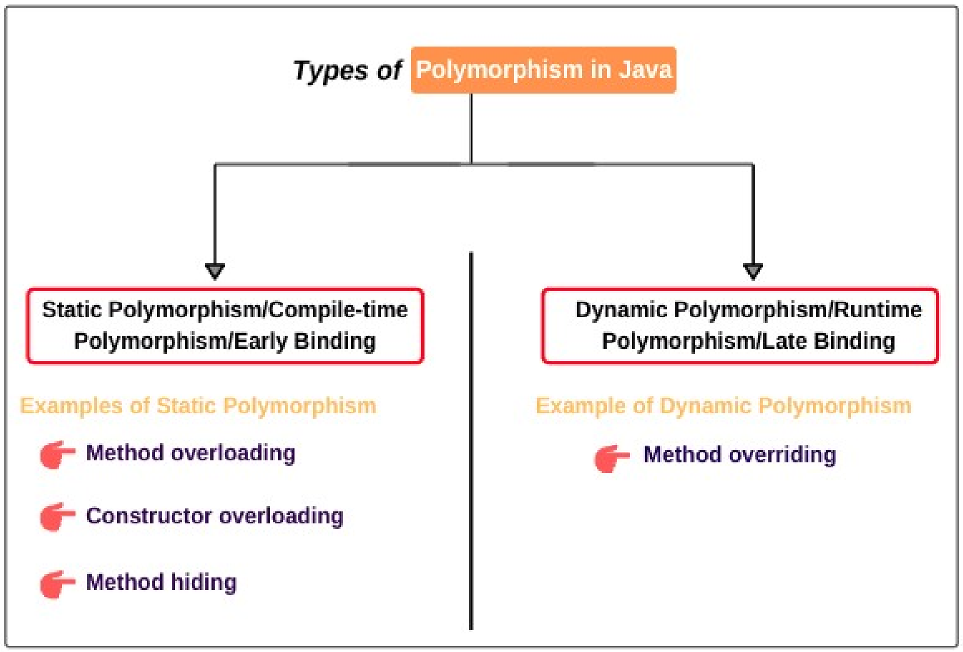
Output

```
Number1=10
```

Number2=20
Number3=30

Polymorphism

- **Polymorphism in Java** is a concept by which we can perform a *single action in different ways*.
- Polymorphism is derived from 2 Greek words: poly and morphs.
- The word "poly" means many and "morphs" means forms. So polymorphism means many forms.


Types of Polymorphism in Java
Static Polymorphism/Compile-time Polymorphism/Early Binding
Dynamic Polymorphism/Runtime Polymorphism/Late Binding
Examples of Static Polymorphism
Method overloading
Constructor overloading
Method hiding
Example of Dynamic Polymorphism
Method overriding

- Static Polymorphism
    - static polymorphism is achieved through method overloading.
    - Method overloading means there are several methods present in a class having the same name but different types/order/number of parameters.
    - At compile time, Java knows which method to invoke by checking the method signatures.
    - So, this is called compile time polymorphism or static binding.
    - Method Overloading on the other hand is a compile time polymorphism-example program.

◦ **Example program**

```
class Overload
 {
        void demo (int a)
        {
        System.out.println ("a: " + a);
        }
        void demo (int a, int b)
         {
        System.out.println ("a and b: " + a + "," + b);
        }
        double demo(double a)
        {
        System.out.println("double a: " + a);
        return a*a;
         }
 }
class Poly1
        {
                public static void main (String args [])
                {
                Overload Obj = new Overload();
                Obj .demo(10);
                Obj .demo(10, 20);
                double result = Obj .demo(5.5);
                System.out.println("O/P : " + result);
                }
        }
```

<u>Output</u>

```
a: 10
a and b: 10,20
double a: 5.5
O/P : 30.25
```

- Here the method demo() is overloaded 3 times:
  - first method has 1 int parameter,
  - second method has 2 int parameters and

    ◦ third one is having double parameter.

- Which method is to be called is determined by the arguments we pass while calling methods.
- This happens at compile time so this type of polymorphism is known as compile time polymorphism.

- **Dynamic Polymorphism in Java**
    - The type of polymorphism which is implemented dynamically when a program being executed is called as dynamic polymorphism.
    - The dynamic polymorphism is also called run-time polymorphism or late binding.
    - Method overriding is one of the ways in which Java supports Runtime Polymorphism.
    - Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
    - In method overriding the method must have the same name as in the parent class and it must have the same parameter as in the parent class.
    - If subclass (child class) has the same method as declared in the parent class, it is known as method overriding in Java.
    - Method overriding occurs in two classes that have inheritance.

- **Dynamic method dispatch**
    - Dynamic method dispatch is a mechanism by which a call to an overridden method is resolved at runtime.
    - This is how java implements runtime polymorphism. When an overridden method is called by a reference, java determines which version of that method to execute based on the type of object it refer to.
    - In simple words the type of object which it referred determines which version of overridden method will be called.
    - It allows subclasses to have common methods and can redefine specific implementation for them.

- ◦ This lets the superclass reference respond differently to same method call depending on which object it is pointing.

- For example,

```
Base b=new Base();
Derived d=new Derived();
b=new Derived();      // Base class reference refer to derived class
                                  // object  that is upcasting
```

- ➢ Here Base is the parent class and b is the object of this class.
    - ➢ Derived is the child class and d is the object of this class.
- ➢ In the third line of the program segment, the parent class reference variable 'b' refers to a child class object than it is called upcasting and using this technique dynamic method dispatch perform.
- ➢ Example program

```
class Rectangle
{
   int l,b;
   int a;
   Rectangle( )
   {
        l=5;
        b=30;
    }
   void area()
   {
       a=l*b;
       System.out.println("Area of rectangle: "+a);
   }
}
class Square extends Rectangle
{
  void area()   //overridden method
  {
   a=l*l;
    System.out.println("Area of square: "+a);
     }
}
class Triangle extends Rectangle
{
    void area()     //overridden method
    {
       a=(b*l)/2;
      System.out.println("Area of triangle: "+a);
     }
}
 class Calculation
{
        public static void main(String args[])
        {
        Rectangle r=new Rectangle();
        r.area();
        r=new Square();  //superclass reference referring to subclass
                         // Square object so,at run time it will call Square
                                 //area()
        r.area();
        r=new Triangle();    //superclass reference referring to subclass T
                                 //riangle object so,at run time it will call triangle
                                 //area()
        r.area();

        }
}
```

Output

```
Area of rectangle: 150
Area of square: 25
Area of triangle: 75
```

- At first, inside of the main method we declared the r as the object of Rectangle class.
- The object r will call the Rectangle class area() method.
- r=**new** Square(); this statement treated r as a reference variable of the class Rectangle and it is converted as an actual object of Square class.

- On the next line where we called the area( ) method on r.
- The java compiler verifies that indeed Rectangle class has a method named area( ), but the java runtime notices that the reference is actually an instance of class Square.
- So it calls Square class area( ) method.
- The same procedure is repeated with Triangle class also.

abstract keyword in java

- 'abstract' keyword is used to declare the method or a class as abstract.
- A class which contains the abstract keyword in its declaration is known as an abstract class.
  - If a class is declared abstract, it cannot be instantiated.
  - An abstract class is a restricted class that cannot be used to create objects.
  - A class which is not abstract is referred as Concrete class.
  - Example,

```
abstract class A
{
        //body of the abstract class
}
```

Here A is an abstract class

- Abstract classes may or may not contain *abstract methods*,
- But, if a class has at least one abstract method, then the class must be declared abstract.
- If we want a class to contain a particular method but the actual implementation of that method to be determined by child classes, then we declare the method in the parent class as an abstract.
- The abstract keyword is used to declare the method as abstract.
- An abstract method contains a method signature, but no method body.
- Instead of curly braces, an abstract method will have a semicolon (;) at the end.
- For example,

```
abstract class Employee
{
        abstract void work( );
}
```

- ⦿ The role of an abstract class is to contain abstract methods. However, it may also contain non-abstract methods. The non-abstract methods are known as concrete methods.

- ⦿ For example,

```
abstract class MyClass
{
public void disp()
{
        System.out.println("Concrete method of parent class");
}
 abstract public void disp2();
}
```

- ✓ **Complete example for abstract**

```
// abstract class
abstract class Shape
{
  // abstract method
  abstract void sides();
}
class Triangle extends Shape
{
  void sides()
  {
    System.out.println("Triangle shape has three sides.");
  }
}
class Pentagon extends Shape
{
  void sides()
  {
    System.out.println("Pentagon shape has five sides.");
  }
}
class abstract_demo
```

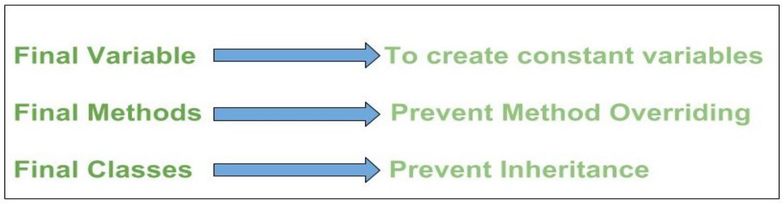
```
{
    public static void main(String[] args)
    {
        Triangle obj1 = new Triangle();
        obj1.sides();
        Pentagon obj2 = new Pentagon();
        obj2.sides();
    }
}
```

Output

```
Triangle shape has three sides.
Pentagon shape has five sides.
```

final Keyword In Java

- All methods and variables can be overridden by default in subclasses.
- Using the final keyword means that the value can't be modified in the future.
- Following are different contexts where final is used.



- **Final variables**
    - When a variable is declared with *final* keyword, its value can't be modified.
    - We must initialize a final variable, otherwise compiler will throw compile-time error.
    - Example,

      final int SIZE=10;
    - The only difference between a normal variable and a final variable is that we can re-assign value to a normal variable but we cannot change the value of a final variable once assigned.
    - Hence final variables must be used only for the values that we want to remain constant throughout the execution of program.

- **Final methods**

- When a method is declared with *final* keyword, it is called a final method.
- A final method cannot be overridden.
- Making a method final ensures that the functionality defined in this method will never be altered in any way.
- Constructors cannot be final.
- Example,

```
final void showData()
{
        ..................
}
```

Final class

- When a class is declared with *final* keyword, it is called a final class.
- A final class cannot be extended(inherited).
    - One is definitely to prevent inheritance, as final classes cannot be extended.
    - For example, all Wrapper Classes like Integer, Float etc. are final classes.
- Example,

```
final class student
{
        ................
}
```

- **Example**

```
final class XYZ
{
        public void display()
        {
                System.out.println("This is a final method.");
        }
 }
class ABC extends XYZ // Compiler Error: cannot inherit from final XYZ
{
        void demo()
        {
                System.out.println("My Method");
        }
}
```